Enhancing Cryptographic Code against Side Channel Cryptanalysis with Aspects

Jérôme Dossogne and Stéphane Fernandes Medeiros

Université Libre de Bruxelles, Boulevard du Triomphe - CP212, 1050, Bruxelles, Belgium

Abstract. In this paper we introduce a new way to protect software implementation of cryptographic protocols against Side Channel Attacks (SCA) using Aspect Oriented Programming (AOP). For this purpose we have implemented the RSA algorithm in Java and our aspects with AspectJ. As a result, we show how AOP can help tremendously to enhance cryptographic protocols against SCA with nearly no negative side-effects. Moreover, we illustrate a new countermeasure against timing attacks aiming for the simple modular exponentiation technique. Our simulation performs a timing attack against the hamming weight of the secret key in a RSA cryptosystem. The success rate of the attack drops from 80% to 0% with our countermeasure.

1 Introduction

In most security protocols, the device running the cryptographic primitives, interacting with the user or with other devices is often viewed, analysed or considered as a black box. Describing protocols [1–4], authors suppose the existence of a device or program able to perform certain operations without leaking information to the outside world. For instance, in [5] writing the following: "*To improve the level of security, a signature* $\sigma = Sign_U(A_U, A_E, M, C, D)$ *... should be considered*", the author make the hypothesis that the key used for the calculation is not leaked, i.e. displayed on the screen, made accessible through the network, ... during the computations.

While most leakage can be prevented by sound implementation of such protocols, sometimes, it remains possible to attack the running system by monitoring its behaviour (e.g. time to compute, ...) or its impact on its surrounding (e.g. thermal energy or other kind of electromagnetic field radiating from the device, ...). These attacks are known as side channel attacks (SCA). Introduced in [6, 7] by Kocher, these attacks can, in some cases, help an attacker recover sections of the cryptographic key used in the cypher of an implemented and running cryptographic protocol.

To secure such implementation from side channel attacks, counter-measures have been proposed [8]. However, most of them require the implementation to be altered. This could be considered as a major drawback slowing down the adoption of such counter-measure and thus preventing the protocols to be securely implemented against SCA's. In some cases, the implementation's source code is certified, has been thoroughly reviewed, is not open source, is under the responsibility of another team, company, ... All these situations could prevent a programmer to enhance the implementation of a protocol in order to protect it against one or different type of SCA.

Dossogne J. and Fernandes Medeiros S..
Enhancing Cryptographic Code Against Side Channel Cryptanalysis With Aspects.
DOI: 10.5220/0003573800390048
In *Proceedings of the 8th International Workshop on Security in Information Systems* (WOSIS-2011), pages 39-48
ISBN: 978-989-8425-61-4

In this paper, we suggest the use of a growing [9] state of the art development technology called Aspect Oriented Programming (AOP) to implement such counter-measures. We show how this methodology could separate the responsibility in the development of the protocol and of the counter-measures, make such development more flexible and even suggest a novel counter-measures that can be easily implemented using AOP.

In section 2 we present various counter-measures and their drawbacks. In section 3, we introduce AOP. In section 4, we present our contribution by introducing how simply AOP can be of used in this context, we illustrate our point with an applied example and show the result of our experiment. We conclude in section 5.

2 Side Channel Attacks and Countermeasures

Early moderns techniques created to attack cryptographic algorithms where based on mathematical properties of those algorithms. These attacks are best known as linear cryptanalysis (discovered in 1992 [10]) and differential cryptanalysis (discovered in the early 90's [11]). Algorithm such as DES [12] and AES [13–15] where later on designed to prevent such attacks [1].

During the second half of the 90's, a new kind of cryptanalysis appeared [6, 7, 17–19] and is nowadays called side channel cryptanalysis. It is based on information leaked by the implementation of the targeted algorithm such as the execution time, the power consumption, the electromagnetic emanations, ...

To avoid leakage and thus to protect the implementation against such attacks, various countermeasures have been proposed such as: blinding, masking [8], hiding [8], random / constant / adaptive idle-wait [20], ...

Blinding [20] is the most usual countermeasure for the algorithm RSA and consists of performing the decryption on a randomised version of the ciphertext. The randomisation has, of course, to be performed in a way that can be reversed after decryption. It has only a small performance penalty. However, as analysed in [6], using blinding techniques is not enough to stop an attack from inferring the Hamming weight of the secret key.

Random idle-wait, constant idle-wait and adaptive idle-wait consist of a family of countermeasure wherein a delay is introduced during the decryption process. These techniques have the advantage of using less power since the processor can be used to a different task while the process is waiting. Adaptive idle is an optimised version of the idle-wait technique with performance oriented design.

More generally, to protect an algorithm against timing attacks, [21] and [22] suggest firstly to ensure that all operations take exactly the same amount of time which should, of course, be independent of the message to encode and from the value(s) of the key(s). Since most countermeasures against SCA of any kind have the same purpose, i.e. to decorrelate the leaked information from the secret information such as the message and the secret key, some of them can work against several SCA. For example, [8]'s suggestions (such as random insertion of dummy operations, random insertion of dummy

[1] As indicated in [16] and contrary to some beliefs, differential cryptanalysis was known before DES was designed which explains its resistance to such attacks.

cycles, skipping clock pulses, randomly changing the clock frequency, multiple clock domains, ...) against power analysis attacks can also be used against timing attacks.

In [6], Kocher introduces timing attacks with the modular exponentiation using the standard simple modular exponentiation algorithm [6]. In that algorithm, the number of modular multiplications is directly correlated to the number of bits set to one in the key. Therefore, the algorithm performs a multiplication each time it encounters a "1". Performing such a multiplication takes a noticeable number of time in regard to the alternative, which is a simple variable assignation when the algorithm encounters the value "0". In the case of the RSA cypher [23], since the values are usually at least 1024 bits long, this factor is even more noticeable. As you will see in section 4, we will illustrate our methodology applying our countermeasure on Kocher's example.

3 About Aspects and AOP

An aspect is a logic scattered or tangled across the code. Such a logic is thus harder to detect, maintain and understand. For instance, an aspect could be a piece of code which was "copied & pasted" into several function, e.g. the call to a method called `checkIntegrity()` at the end of each method of an abstract data type developed with the design by contract methodology [24] could be considered as an aspect. While such a code could make a lot of sense, it usually does not fit with the logic and purpose of the methods of the class that calls it.

Aspect-oriented programming (AOP) is a programming paradigm whose purpose is to isolate secondary or supporting functions from the main program's core logic. AOP allows the programmer to externalise cross-cutting concerns, the aspect(s), and thus clean the class's code which afterwards only contains code regarding its core function. Using what is called an Aspect weaver, it is thus possible to alter, dynamically or not, the original code with external specification without manually entangle the modification in the code itself and thus without mixing nor the logic of the class and of the aspect nor the programmers responsibilities for these two codes. Lots of languages have implementation of AOP such as C++, PHP, Delphi, Smalltalk, Java, ... within the language or as an external library. The amount of features and possibilities depend on the weaver's implementation.

In this paper, we will illustrate our point and the AOP model (also called Join Point[2] Model (JPM)) using AspectJ, a Java implementation of AOP. To the author's best knowledge it is the most popular and widely known general purpose Aspect weaver.

3.1 The Dynamic Join Point Model of AspectJ

[3] It is made of 4 components: poincuts, advice, inter-type declaration and aspects.

[2] A join point is a point of the program that an aspect could impact. It can be structural (classes, methods, ...) or behavioural (call to a method, modification of variable, ...).

[3] This section was inspired by www.eclipse.org's presentation of AspectJ's model [25].

Pointcuts. A pointcut designate a set of join points in the program execution flow. For instance, `call(void Point.setX(int))` designate every join points in the flow where a method having the signature `void Point.setX(int)` is called. AspectJ allows the programmer to name these pointcuts and to combine them using logical operator ”and”, ”or” and ”not”.

Advice. An advice defines the crosscutting behaviour’s code. Once the pointcuts are defined, advices can be associated with these pointcuts using keywords such as ”before”, ”after”, ”after returning”, . . . which allows the programmer to indicate when exactly he wishes the aspect to be executed. During the execution of an advice, the latter can be made aware of its execution context by the pointcuts. AspectJ’s advice operate essentially dynamically.

Inter-type Declarations. AspectJ’s inter-type declarations cut across classes and their hierarchies. For instance, they allow the programmer to alter the content or even the inheritance relationships of multiple classes. AspectJ’s inter-type declarations operate at compile-time.

Aspects. An aspect in AOP is basically the equivalent of the classes in Object Oriented Programming as it can own methods, fields and initialisers in addition to the crosscutting members (pointcuts, advice and inter-type declarations) that they regroup.

4 Implementing Countermeasures with AOP

“AspectJ lean modularisation of crosscutting concerns, such as error checking and handling, synchronisation, context-sensitive behaviour, performance optimisations, monitoring and logging, debugging support, and multi-object protocols”(AspectJ’s homepage: eclipse.org/aspectj/)

In this section we propose a novel way of using AOP that, as far as we know, has not yet been studied (especially the use we are proposing is not considered in [25] nor in [9]). Specifically we propose the use of AOP in the context of the implementation of countermeasures against some side channel attacks. In this paper we outline the first ideas that would allow to further develop this new defence technology.

In section 2, we evoked several countermeasures, namely: blinding, random / constant / adaptive idle-wait, random insertion of dummy operations, random insertion of dummy cycles, skipping clock pulses, randomly changing the clock frequency, multiple clock domains, . . . Most of those countermeasures and more can be implemented via aspects.

To blind the message, one could place a pointcut on the execution of the encryption and decryption method[4]. The advice could then substitute the value of the message by its blinded counterpart.

[4] In a public key cryptosystem, one could argue the need to enhance both the encryption and decryption process since one of the two keys is always public.

The family of idle-wait countermeasures could be implemented with a pointcut on the access to the object's attribute or on the call/execution of the encryption or decryption method. The advice would be a simple `sleep()` call with a parameter depending on the member of the family.

The "random insertion of dummy operations" can also be implemented with the same pointcuts and with adequate advices containing the dummy operations in question.

The same can be said for the last countermeasures. Notice that all of these operations do not have to be deterministic since an advice can have a probabilistic behaviour. Moreover, the most trivial countermeasure could be to use aspects to intercept the call to the encryption process and substitute the algorithm with another which could produce the same results and be time independent.

We will illustrate below such an implementation on the RSA cryptosystem with a countermeasure that we created for the standard simple modular exponentiation technique (SSMET) [6]. As indicated, the source code contains other implementations of the modular exponentiation algorithm. Switching from the most naive algorithm to the one used to illustrate our point is already a countermeasure since some others[5] leak easily the value of the key while the second is faster and only leaks the hamming weight of the secret key. We then implement a countermeasure to stop such that leak.

4.1 The RSA Cryptosystem

The RSA cryptosystem was first described by Ron Rivest, Adi Shamir and Lan Adleman in 1977 [26]. It is an algorithm for public-key cryptography [27] used for encryption and digital signatures.

Mathematical Principles. The private (e) and public (d, n) keys are generated following equations 1 and 2 and, with the message m, the encryption and decryption procedure is illustrated with equation 3.

$$n = pq,\ 0 < p < q < n : n, p, q \in \mathbb{Z} \tag{1}$$

$$\phi(n) = (p-1)(q-1),\ ed = 1 \bmod \phi(n),\ e < \phi(n) < n \tag{2}$$

$$m \in \mathbb{Z}_n, c = m^e \bmod n, m = c^d \bmod n \tag{3}$$

There exist several ways to compute the modular exponentiation: the naive algorithm ($a^n = \prod_{i=1}^{n} a$), square-and-multiply algorithm [28], via the explicit Chinese remainder theorem [29], SSMET [6], ...

Our Simulation. After the creation of thousands of keys using our `RSAKeyGenerator` class, we simulated series of encryption and decryption on a randomly chosen message. The encryption and decryption process uses the SSMET. With x the key, n the modulus, w its size and considering bits from the most significant bit (MSB) to the least significant bit (LSB), the algorithm looks like this:

[5] Other techniques are implemented and available in the source code for testing purpose.

```
Let s_0 = 1.
For k = 0 upto w-1:
    If (bit k of x) is 1 then
        Let R_k = (s_k . y) mod n.
    Else
        Let R_k = s_k
    Let s_{k+1} = R_k^2 mod n.
EndFor.
Return (R_{w-1}).
```

The simulation's full code is available online [30]. It is *not* intended for real world application. The purpose of this paper is not to present an efficient and resistant implementation of RSA nor to present an intelligent countermeasure against timing attack for the RSA cypher. The code purpose is to demonstrate how easily AOP could be used to implement some countermeasures without the usual difficulties and mixed responsibilities. We implemented a cypher with the intent of illustrating a timing attack. It's running time is dependant on key's hamming weight on purpose. Therefore a constant delay is performed after each multiplication to simulate the ability of an attacker to measure the time with more precision than what the Java Virtual Machine offers with its `System.currentTimeMillis()` method[6]. The code can be easily modified to use a real world implementation of the function as we included the corresponding less experimental code from the BigInteger library in the release source code. The key generator can be reused but the selected e should be chosen at random[7] and the default key size should be modified from 256 bits to at least 1024, 2048 bits or 4096 bits as recommended for real world application [31].

4.2 A Timing Attack

Without considering side channel cryptanalysis, the security of RSA is based on the difficulty to factorise large numbers. Several attacks are presented in [22]: common modulus attack, low private exponent attack, low public exponent, Hastad's broadcast attack,... However, these attacks are mostly effective when the algorithm is misused.

As explained in section 2, in the SSMET, since a multiplication is performed only when the algorithm encounter a bit set to "1" in the binary representation of the encryption or decryption key, the time it takes to perform its task is directly proportional to the key's hamming weight. Thus, by measuring that time, the attacker obtains a leaked information. Such an information can drastically increase the speed of an attack against the scheme since, for instance, only the keys with such a hamming weight would have to be tested in a brute force attack. Therefore, if the key size is 1024 bits and has a hamming weight of 1023, instead of having roughly 2^{1024} possible keys to try, the attacker only has to try $\binom{1024}{1023} = \frac{1024!}{1023!1!} = 1024$ keys.

This attack's success rate is inversely proportional to the desired precision of the

[6] Which seems a fairly reasonable hypothesis.

[7] We currently start searching for possible values for e from a probable random prime of $(\text{key size})/2$ bits which speeds up the process but is not mandatory and thus discard potential values for e.

hamming weight's approximation and proportional to the attackers measuring capabilities (simulated here by the magnitude of the induced delay in the multiplication), e.g. the success rate will be lower if the delay is shorter, the key size is longer and if the approximation needs to be very precise[8]. More clever attacks would perform multiple decryption using potentially different messages and try to guess the hamming weight using advanced statistical or machine learning techniques whereas, here, no technique is used, only one measurement on a single message.

4.3 A Countermeasure

As explained in section 2, our countermeasure will simply ensure that the encryption or decryption process always takes a fixed and key independent amount of time since our purpose, in this example, is to protect the secret key.

Another approach could be to add a random delay to each call to the exponentiation function. However, while this is out of the scope of this paper, we are afraid that the latest, a noise, could easily be defeated by averring the time taken to decrypt the same message with the same key and thus we did not use it to illustrate our point.

Our Aspect. The easiest way to implement such a countermeasure is to modify the source code directly in order to perform a multiplication whether the encountered key bit's value is "1" or not. As previously explained, this leads to mixing the responsibility of the programmers responsible for the implementation of a correct and sound cypher and of the programmers responsible of enhancing that implementation against contextual side channel attacks. Moreover, this leads to a more complicated code and thus the reviewing of such a code would also be more complicated. Furthermore, if a reviewing committee has declared the code as "correct", no more alteration could be allowed since the whole code would have to be reevaluated.

To escape such a fate, we will use AOP and design an aspect that could be reviewed and evaluated separately from the original source code. The downfall of this technique would be to trust the aspect weaver in addition to the previous trust already given to the Java environment.

Our salvation will come from a simple aspect. It will execute the computations of an encryption process on the same message but with the binary complement, $\overline{e}$, of the secret key e then proceeds with the requested encryption. Thus, if the Hamming weight (HW) of the key e is x ($HW(e) = x$) and the key is stored with w bits, instead of performing x multiplication during the encryption process, we will always perform w multiplication (since $HW(\overline{e}) = w - x$ and $w - x + x = w$). This aspect implements a new kind of countermeasure close to the idle family, but different from the definition of constant, random or adaptive idle. Indeed, firstly the computer is never really idle and secondly the execution time is constant relatively to the system parameters, i.e. the size of the potential keys. It corresponds to maxTime, the maximum amount of time that the modular exponentiation function could take for any key, i.e. the required time is the key is made solely of 1's.

[8] All these parameters are taken into account in our simulation and can be modified at will.

```
public aspect PowerComplement {
   long around(long n, long e, long m) :
    (execution(* RSA.decrypt(..))) &&
    args(n,e,m) {
      RSA.powerModHamming(n,RSA.getMaxValue().subtract(e),m);
      return proceed(n, e, m);
   }
}
```

The aspect `PowerComplement` is made of (1) the pointcuts `long around(long n, long e, long m)`, which substitutes any execution of the method `decrypt` of the class `RSA` and with three parameters (n, e, m) with

If one wishes to implement a constant idle countermeasure, as described in section 2, after defining a constant $t >$ maxTime, replacing the body of the previous aspect by the following should be enough:

```
      long start = System.currentTimeMillis();
      long res = RSA.powerModHamming(n,e,m);
      long stop = System.currentTimeMillis();
      Thread.sleep(t-(start-stop));
      return res;
```


4.4 Results

The experimentation's sets of keys are made respectively of 20, 100, 1 000, 10 000 and 100 000 distinct keys generated by our key generator. For each launch, a random message $m \in \mathbb{Z}_n$ is chosen. The launch proceeds by encrypting and decrypting that message with all the keys in the set. Measurements are taken before and after the deciphering and are saved. Using those measurement, the side-channel attack is launched and a record is kept of each correct guess of the key's hamming weight. The procedure is exactly the same during the experimentation with the countermeasure. The success rate of the attack increases if the delay introduced for each multiplication is increase, which simulates the ability to measure that particular moment or to suppress the influence of the rest of the code with a greater precision. To break the suspense, without countermeasure and independently of the chosen message or set of keys, the success rate of the attack for keys of 256 bits were always between 80% and 84%. With the countermeasure, the success rate of the attack is 0% whatever the delay, the message, the precision or the key size were. These sets of keys, measurements and statistics are available at [30].

5 Conclusions

In this paper we suggested a new domain of applications for aspect oriented programming, naming the flexible implementation of countermeasure against side-channel attacks. We introduced the notions of aspect and side channel cryptography before presenting the application and illustrated the latest with an experiment.

The first result is the obvious easiness to create and implement countermeasure against side channel attacks (SCA) without altering the original code of the cryptosystem thanks to aspects. Indeed, all of the previously mentioned drawbacks disappeared with, however, the cost of trusting the aspect weaver. This methodology is thus well indicated in order to enhance existing cryptographic code without the need to edit the latest. This leaves us with a brand new field of experimentation. For instance, we intend to explore the domain of Smart Card and the possibility to reinforce existing cryptographic code using aspects where it is needed and possible.

The second is the perfect effectiveness of the new proposed countermeasure against timing attacks for the simulated RSA cryptosystem.

We wish to stress the following point: In order to implement countermeasure against SCA's, it is obvious that the programmer has to take the context in consideration before choosing his language and then the most adequate aspect weaver corresponding to that language. The choice we made to illustrate our point via an experiment using Java and AspectJ was based, as explained on the weaver's capacities to suit our needs for this paper. It was in no case a suggestion to use that language and weaver in every context. Also, one has to keep in mind that implementing a countermeasure to protect against a certain kind of SCA can sometimes introduce a leak of information that could be captured by an other kind of SCA [32].

References

1. Dossogne, J., Markowitch, O.: E-voting : Individual verifiability of public boards made more achievable. In Goseling, J., Weber, J.H., eds.: Proceedings of the 31th Symposium on Information Theory in the Benelux (WICSITB2010), Rotterdam, The Netherlands (2010) 5–10
2. Dossogne, J., Markowitch, O.: Voting With a Tripartite Designated Verifier Scheme Based On Threshold RSA Signatures. In Tjalling, T., Willens, F., eds.: Proceedings of the 30th Symposium on Information Theory in the Benelux (WIC09). Volume 1., Eindhoven (2009) 113–118
3. Desmedt, Y., Elkind, E.: Equilibria of plurality voting with abstentions. In: Proceedings of the 11th ACM conference on Electronic commerce - EC '10, New York, New York, USA, ACM Press (2010) 347
4. Kremer, S., Ryan, M., Smyth, B.: Election verifiability in electronic voting protocols. In Gritzalis, D., Preneel, B., Theoharidou, M., eds.: Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS'10). Volume 6345 of Lecture Notes in Computer Science., Athens, Greece, Springer-Verlag (2010) 389–404
5. Dossogne, J., Markowitch, O.: Online banking and man in the browser attacks , survey of the belgian situation. In Goseling, J., Weber, J.H., eds.: Proceedings of the 31th Symposium on Information Theory in the Benelux (WICSITB2010), Rotterdam, The Netherlands (2010) 19–26
6. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Koblitz, N., ed.: CRYPTO'96. Volume 1109 of Lecture Notes in Computer Science., Springer (1996) 104–113
7. Kocher, P. C., Jaffe, J., Jun, B.: Differential Power Analysis. In Wiener, M. J., ed.: Advances in Cryptology - CRYPTO99. Volume 1666 of Lecture Notes in Computer Science., Springer (1999) 388–397

8. Mangard, S., Oswald, M. E., Popp, T.: Power Analysis Attacks - Revealing the Secrets of Smart Cards. Springer (2007)
9. Sertkaya, Y.: Application Areas of Aspect Oriented Programming (2009)
10. Matsui, M., Yamagishi, A.: A New Method for Known Plaintext Attack of FEAL Cipher. In Rueppel, R.A., ed.: Advances in Cryptology EUROCRYPT92. Volume 658 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer Berlin Heidelberg (1993) 81–91
11. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. Journal of Cryptology 4 (1991) 3–72
12. Federal Information/NIST: DATA ENCRYPTION STANDARD (DES) (Processing Standards Publication 46-2) (1999)
13. National Institute of Standards and Technology (NIST): AES Algorithm (Rijndael) Information (2001)
14. Daemen, J., Rijmen, V.: AES submission document on Rijndael (amended) (2003)
15. National Institute of Standards and Technology (U.S. Department of Commerce): FIPS PUB 197, Advanced Encryption Standard (AES) (2001)
16. Coppersmith, D.: The Data Encryption Standard (DES) and its strength against attacks. IBM Journal of Research and Development 38 (1994) 243–250
17. Bernstein, D.J.: Cache-timing attacks on AES (2005)
18. Yang, B., Wu, K., Karri, R.: Scan Based Side Channel Attack on Data Encryption Standard (Cryptology ePrint Archive, Report 2004/083) (2004)
19. Kühn, U.: Side-Channel Attacks on Textbook RSA and ElGamal Encryption. In Desmedt, Y.G., ed.: Public Key Cryptography PKC 2003. Volume 2567 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer Berlin Heidelberg (2002) 324–336
20. Moreno, C., Hasan, M. A.: An Adaptive Idle-Wait Countermeasure Against Timing Attacks on Public-Key Cryptosystems (2010)
21. Shamir, A.: Method and apparatus for protecting public key schemes from timing and fault attacks (1999)
22. Boneh, D.: Twenty Years of Attacks on the RSA Cryptosystem (1999)
23. RSA Security Inc.: RSA Raw Encryption using the JCE (2011)
24. McCafferty, B.: Design-by-Contract: A Practical Introduction (2006)
25. Xerox Corporation, Palo Alto Research Center: Introduction to AspectJ (2011)
26. Rivest, R. L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21 (1978) 120–126
27. RSA Laboratories: What is public-key cryptography? (2010)
28. Dhem, J. F., Koeune, F., Leroux, P. A., Mestré, P., Quisquater, J. J., Willems, J. L.: A Practical Implementation of the Timing Attack. In Quisquater, J.J., Schneier, B., eds.: Smart Card. Research and Applications Third International Conference, CARDIS98. Volume 1820 of Lecture Notes in Computer Science., Louvain-la-Neuve, Belgium, Springer (1998) 167–182
29. Bernstein, D. J., Sorenson, J. P.: Modular exponentiation via the explicit Chinese remainder theorem. Mathematics of Computation 76 (2007) 443–455
30. Dossogne, J.: homepages.ulb.ac.be/˜jdossogn (2011)
31. RSA Security Inc.: RSA Laboratories - 4.1.2.1 What key size should be used? (2011)
32. Yen, S. M., Kim, S., Lim, S., Moon, S. J.: A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In Kim, K., ed.: Information Security and Cryptology ICISC 2001. Volume 2288/2002 of Lecture Notes in Computer Science., Springer (2002) 269–294